Using **Physics-Informed Neural Networks (PINNs)** to solve the **Navier-Stokes equations with astrophysical flow considerations**

ISHAN TANDON

INDIAN INSTITUTE OF TECHNOLOGY, ROORKEE PAAC OPEN PROJECT 2024

Instructors: *Ms. G. Jeevika, Mr. Adith V R, Ms. Khushi* Date: 8 December, 2024

ABSTRACT

Differential equations are omnipresent in our world. Every phenomenon, ranging from heat transfer, fluid mechanics to predicting fields in astrophysics, can be modeled with the help of differential equations. Being able to produce exact solutions of these differential equations turns out to be of extreme importance in every field.

Classically, there are two ways to solve differential equations, analytical and numerical. Analytical methods try to find the exact solutions to differential equations. Unfortunately, these methods are limited to a limited number of differential equations. For others (i.e., for most), we have to employ the use of Numerical Methods which try to 'approximate' the solutions of these differential equations. Sadly, the ability of these numerical methods to find the solutions depend upon the computational capability of the underlying system and are prone to errors in high-dimensional or dynamic systems.

This is where PINNs step it. Physics-informed neural networks (PINNs) offer a promising alternative by embedding the governing physical laws directly into the neural network's loss function. This approach enables the model to approximate solutions that obey the underlying physics without relying on extensive data or complex meshing. They can be employed in situations where the computation demand would be high for traditional numerical methods or where data is scarce.

In this project, we aim to demonstrate the effectiveness of Physics-Informed Neural Networks (PINNs) in solving the Navier-Stokes equations for astrophysical flows. The Navier-Stokes equations, fundamental to fluid dynamics, describe the motion of fluid substances and play a crucial role in modeling phenomena like accretion disks, stellar winds, and interstellar medium dynamics in astrophysics. These equations are challenging to solve due to their non-linearity and the complexity introduced by astrophysical boundary conditions.

We incorporate relevant astrophysical parameters such as velocity fields, pressure, magnetic field components, and their interactions to accurately model these flows. By leveraging PINNs, which integrate physical laws into the learning process, the model ensures compliance with governing equations while utilizing neural networks' flexibility and computational efficiency.

- **Momentum Equation (Navier-Stokes)**:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \frac{1}{\mu_0 \rho}(\nabla \times \mathbf{B}) \times \mathbf{B}$$

where $\mathbf{u}$ is the velocity field, p is the pressure, ρ is the fluid density, ν is the kinematic viscosity, and $\mathbf{B}$ is the magnetic field.

Induction Equation (Magnetic Field Evolution):

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{u} \times \mathbf{B}) + \eta \nabla^2 \mathbf{B}$$

where η is the magnetic diffusivity.

Continuity Equation (for mass conservation):

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0$$

BOUNDARY CONDITIONS

1. No-Slip Boundary Condition (Velocity Constraints)

$u(x,y,t)=0$, $v(x,y,t)=0$ on the domain edges

2. **Periodic Boundary Condition (Magnetic Field Constraints)**

$Bx(x=0,y,t) = Bx(x=1,y,t)$, $By(x,y=0,t) = By(x,y=1,t)$

3.Initial Condition (Time t=0)

- The initial conditions defined the state of the system at the start of the simulation (t=0):

$$u(x, y, t = 0) = \sin(2\pi x)\cos(2\pi y), \quad v(x, y, t = 0) = -\cos(2\pi x)\sin(2\pi y),$$

$$B_x(x, y, t = 0) = \cos(2\pi x), \quad B_y(x, y, t = 0) = \sin(2\pi y).$$

PINN Design

After extensive experimentation and tuning, we developed an optimal neural network architecture for our Physics-Informed Neural Network (PINN) to solve the Navier-Stokes equations for astrophysical flows. Here's the design breakdown:

1. Input Layer
    - The input layer takes three inputs: spatial coordinates x, y, and time t. These inputs are concatenated into a single tensor and passed to the network for feature extraction.

```
x_train = X_train[:, 0:1]
y_train = X_train[:, 1:2]
t_train = X_train[:, 2:3]
```

2. Hidden Layers
    - **First Hidden Layer:** A fully connected (dense) layer that maps the three inputs (x,y,tx,y,t) to 64 neurons. This initial mapping allows the network to start capturing spatial and temporal patterns.
    - **Intermediate Layers:** The network has three additional hidden layers, each with 64 neurons. These layers increase the model's capacity to learn the underlying physics of the Navier-Stokes equations.
    - **Activation Function:** The **tanh** activation function is used for all hidden layers. This choice ensures smooth gradients and helps the network capture the continuous nature of the solution.

```
class PINN(tf.keras.Model):
    def __init__(self, layers):
        super(PINN, self).__init__()
        self.layers_list = [tf.keras.layers.Dense(layer, activation='tanh') for layer in layers[:-1]]
        self.output_layer = tf.keras.layers.Dense(layers[-1], activation=None)

    def call(self, inputs):
        x = inputs
        for layer in self.layers_list:
            x = layer(x)
        return self.output_layer(x)
```

```
layers = [3, 64, 64, 64, 64, 5]
pinn_model = PINN(layers)
```

3. Output Layer
    - The output layer consists of five neurons, representing the predicted values for:
        - u: x-component of velocity.
        - v: y-component of velocity.
        - p: pressure.
        - Bx: x-component of the magnetic field.
        - By: y-component of the magnetic field.
    - No activation function is applied in the output layer to allow the predictions to take on a wide range of values, matching the physical properties of the system.

4. Loss Function Design
    - The PINN integrates the governing Navier-Stokes equations and boundary conditions into the loss function. This ensures the model not only fits the training data but also adheres to the physical laws governing the system.

    - The loss components include:
        - **Physics Loss:** Captures the error in satisfying the Navier-Stokes equations, continuity equations, and induction equations.

```
@tf.function
def loss_fn(model, x, y, t):
    nu = 0.01  # Kinematic viscosity

    with tf.GradientTape(persistent=True) as tape:
        tape.watch([x, y, t])
        inputs = tf.concat([x, y, t], axis=1)
        outputs = model(inputs)
        u, v, p, B_x, B_y = outputs[:, 0:1], outputs[:, 1:2], outputs[:, 2:3], outputs[:, 3:4], outputs[:, 4:5]

        # First-order derivatives
        u_x = tape.gradient(u, x)
        u_y = tape.gradient(u, y)
        v_x = tape.gradient(v, x)
        v_y = tape.gradient(v, y)
        p_x = tape.gradient(p, x)
        p_y = tape.gradient(p, y)
        B_x_t = tape.gradient(B_x, t)
        B_y_t = tape.gradient(B_y, t)

        # Second-order derivatives
        u_xx = tape.gradient(u_x, x)
        u_yy = tape.gradient(u_y, y)
        v_xx = tape.gradient(v_x, x)
        v_yy = tape.gradient(v_y, y)

    del tape  # Free resources

    # Continuity equation
    continuity = u_x + v_y

    # Navier-Stokes equations
    momentum_x = u * u_x + v * u_y + p_x - nu * (u_xx + u_yy)
    momentum_y = u * v_x + v * v_y + p_y - nu * (v_xx + v_yy)

    # Induction equations
    induction_x = B_x_t - (u * B_x + v * B_y)
    induction_y = B_y_t - (u * B_y + v * B_x)
```

```
return tf.reduce_mean(tf.square(continuity)) + \
       tf.reduce_mean(tf.square(momentum_x)) + \
       tf.reduce_mean(tf.square(momentum_y)) + \
       tf.reduce_mean(tf.square(induction_x)) + \
       tf.reduce_mean(tf.square(induction_y))
```

- **Boundary Condition Loss:** Enforces no-slip and periodic boundary conditions.

```
def boundary_condition_loss(model, x_boundary, y_boundary, t_boundary, u_boundary, v_boundary, periodic_boundary):
    inputs_boundary = tf.concat([x_boundary, y_boundary, t_boundary], axis=1)
    outputs_boundary = model(inputs_boundary)
    u_pred, v_pred = outputs_boundary[:, 0:1], outputs_boundary[:, 1:2]

    no_slip_loss = tf.reduce_mean(tf.square(u_pred - u_boundary)) + \
                   tf.reduce_mean(tf.square(v_pred - v_boundary))

    x_left, x_right, y_bottom, y_top = periodic_boundary
    inputs_periodic_x = tf.concat([x_left, y_boundary, t_boundary], axis=1)
    inputs_periodic_x_shift = tf.concat([x_right, y_boundary, t_boundary], axis=1)
    inputs_periodic_y = tf.concat([x_boundary, y_bottom, t_boundary], axis=1)
    inputs_periodic_y_shift = tf.concat([x_boundary, y_top, t_boundary], axis=1)

    outputs_left = model(inputs_periodic_x)
    outputs_right = model(inputs_periodic_x_shift)
    outputs_bottom = model(inputs_periodic_y)
    outputs_top = model(inputs_periodic_y_shift)

    Bx_left, Bx_right = outputs_left[:, 3:4], outputs_right[:, 3:4]
    By_bottom, By_top = outputs_bottom[:, 4:5], outputs_top[:, 4:5]

    periodic_loss = tf.reduce_mean(tf.square(Bx_left - Bx_right)) + \
                    tf.reduce_mean(tf.square(By_bottom - By_top))

    return no_slip_loss + periodic_loss
```

- **Initial Condition Loss:** Ensures the model aligns with the predefined initial velocity and magnetic field values.

```
@tf.function
def total_loss_fn(model, x, y, t, x_boundary, y_boundary, t_boundary,
                  u_boundary, v_boundary, periodic_boundary, x_init, y_init, t_init,
                  u_init, v_init, Bx_init, By_init):
    physics_loss = loss_fn(model, x, y, t)
    bc_loss = boundary_condition_loss(model, x_boundary, y_boundary, t_boundary,
                                      u_boundary, v_boundary, periodic_boundary)
    inputs_init = tf.concat([x_init, y_init, t_init], axis=1)
    outputs_init = model(inputs_init)
    u_pred_init, v_pred_init, Bx_pred_init, By_pred_init = outputs_init[:, 0:1], outputs_init[:, 1:2], outputs_init[:, 3:4], outputs_init[:, 4:5]
    ic_loss = tf.reduce_mean(tf.square(u_pred_init - u_init)) + \
              tf.reduce_mean(tf.square(v_pred_init - v_init)) + \
              tf.reduce_mean(tf.square(Bx_pred_init - Bx_init)) + \
              tf.reduce_mean(tf.square(By_pred_init - By_init))
    return physics_loss + bc_loss + ic_loss
```

5. Optimization
    - The PINN is trained using the Adam optimizer, with a learning rate of 0.001. This optimizer balances stability and convergence speed during training.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
```

6. Epochs : 2000
   We may increase the epochs to increase the accuracy and reduce the loss.

Implementation

Tools Used:

1. Google Colab T4GPU

Software Libraries Used**:**

1. Pytorch
2. Matplotlib
3. Numpy
4. Pandas
5. SkLearn
6. Scipy

GENERATING THE DATASET USING ANALYTICAL METHOD

```
import numpy as np
import matplotlib.pyplot as plt

# Define domain and grid resolution
Nx, Ny, Nt = 10, 10, 10
x = np.linspace(0, 1, Nx)
y = np.linspace(0, 1, Ny)
t = np.linspace(0, 1, Nt)
X, Y, T = np.meshgrid(x, y, t, indexing='ij')

# Parameters for analytical solution
nu = 0.01  # Kinematic viscosity
k = 2 * np.pi  # Wave number
omega = np.pi  # Frequency for time dependence

# Analytical velocity field
def analytical_velocity(x, y, t):
    u = np.sin(k * x) * np.cos(k * y) * np.cos(omega * t)
    v = -np.cos(k * x) * np.sin(k * y) * np.cos(omega * t)
    return u, v

# Analytical pressure field
def analytical_pressure(x, y, t):
    p = -0.5 * (np.sin(k * x) ** 2 + np.sin(k * y) ** 2)
    return p

# Analytical magnetic field
def analytical_magnetic_field(x, y, t):
    Bx = np.cos(k * x) * np.cos(k * y) * np.cos(omega * t)
    By = -np.sin(k * x) * np.sin(k * y) * np.cos(omega * t)
    return Bx, By

# Generate analytical solutions
u, v = analytical_velocity(X, Y, T)
p = analytical_pressure(X, Y, T)
Bx, By = analytical_magnetic_field(X, Y, T)
```

```
    analytical_data = np.stack([X.flatten(), Y.flatten(), T.flatten(),
                                u.flatten(), v.flatten(), p.flatten(),
                                Bx.flatten(), By.flatten()], axis=1)

    # Split into training and testing sets
    np.random.shuffle(analytical_data)
    split_ratio = 0.8
    split_index = int(split_ratio * analytical_data.shape[0])

    train_data = analytical_data[:split_index]
    test_data = analytical_data[split_index:]

    # Save data for later use
    np.save('train_data.npy', train_data)
    np.save('test_data.npy', test_data)
```

INITIALISATION OF DATA

1. Training Data (X_train, Y_train, T_train)

These points are used in the physics-based loss function, which calculates residuals for continuity, momentum, and induction equations.

2. Boundary Conditions (X_boundary, Y_boundary, T_boundary)

The boundary dataset is used to compute the boundary condition loss, enforcing constraints.Boundary conditions define the system's behavior at the edges of the domain, ensuring the solution is physically meaningful and respects known constraints like:

- No-slip condition**:** Velocity is zero at the boundary.
- Magnetic boundary conditions**:** Ensure consistency in magnetic field values at periodic or fixed boundaries.

3. Initial Conditions (X_init, Y_init, T_init)

The initial condition dataset captures the state of the system at the beginning of the simulation (t=0).

4. Periodic Boundary Conditions

Periodic boundary conditions occur when the domain wraps around, such that the solution on one edge of the domain matches the solution on the opposite edge.
For instance, in a 2D periodic domain:

- At x=0, the solution must match x=1.
- At y=0, the solution must match y=1.

Using **Physics-Informed Neural Networks (PINNs)** to solve the **Navier-Stokes equations with astrophysical flow considerations**

```
split_ratio = 0.8
split_index = int(split_ratio * analytical_data.shape[0])

X_train, X_boundary = X[:split_index], X[split_index:]
Y_train, Y_boundary = Y[:split_index], Y[split_index:]
T_train, T_boundary = T[:split_index], T[split_index:]
u_train, u_boundary = u[:split_index], u[split_index:]
v_train, v_boundary = v[:split_index], v[split_index:]
p_train, p_boundary = p[:split_index], p[split_index:]
Bx_train, Bx_boundary = Bx[:split_index], Bx[split_index:]
By_train, By_boundary = By[:split_index], By[split_index:]

# Create initial condition dataset (t=0 slice)
initial_condition_mask = (T.flatten() == 0)
X_init = X[initial_condition_mask]
Y_init = Y[initial_condition_mask]
T_init = T[initial_condition_mask]
u_init = u[initial_condition_mask]
v_init = v[initial_condition_mask]
Bx_init = Bx[initial_condition_mask]
By_init = By[initial_condition_mask]

# Convert data to TensorFlow tensors
x_train, y_train, t_train = map(lambda x: tf.constant(x, dtype=tf.float32), [X_train, Y_train, T_train])
x_boundary, y_boundary, t_boundary = map(lambda x: tf.constant(x, dtype=tf.float32), [X_boundary, Y_boundary, T_boundary])
u_boundary, v_boundary = map(lambda x: tf.constant(x, dtype=tf.float32), [u_boundary, v_boundary])

x_init, y_init, t_init = map(lambda x: tf.constant(x, dtype=tf.float32), [X_init, Y_init, T_init])
u_init, v_init, Bx_init, By_init = map(lambda x: tf.constant(x, dtype=tf.float32), [u_init, v_init, Bx_init, By_init])

# Define periodic boundary data
x_left = tf.constant(np.zeros_like(x_boundary), dtype=tf.float32)
x_right = tf.constant(np.ones_like(x_boundary), dtype=tf.float32)
y_bottom = tf.constant(np.zeros_like(y_boundary), dtype=tf.float32)
y_top = tf.constant(np.ones_like(y_boundary), dtype=tf.float32)
periodic_boundary = (x_left, x_right, y_bottom, y_top)
```

TRAINING LOOP

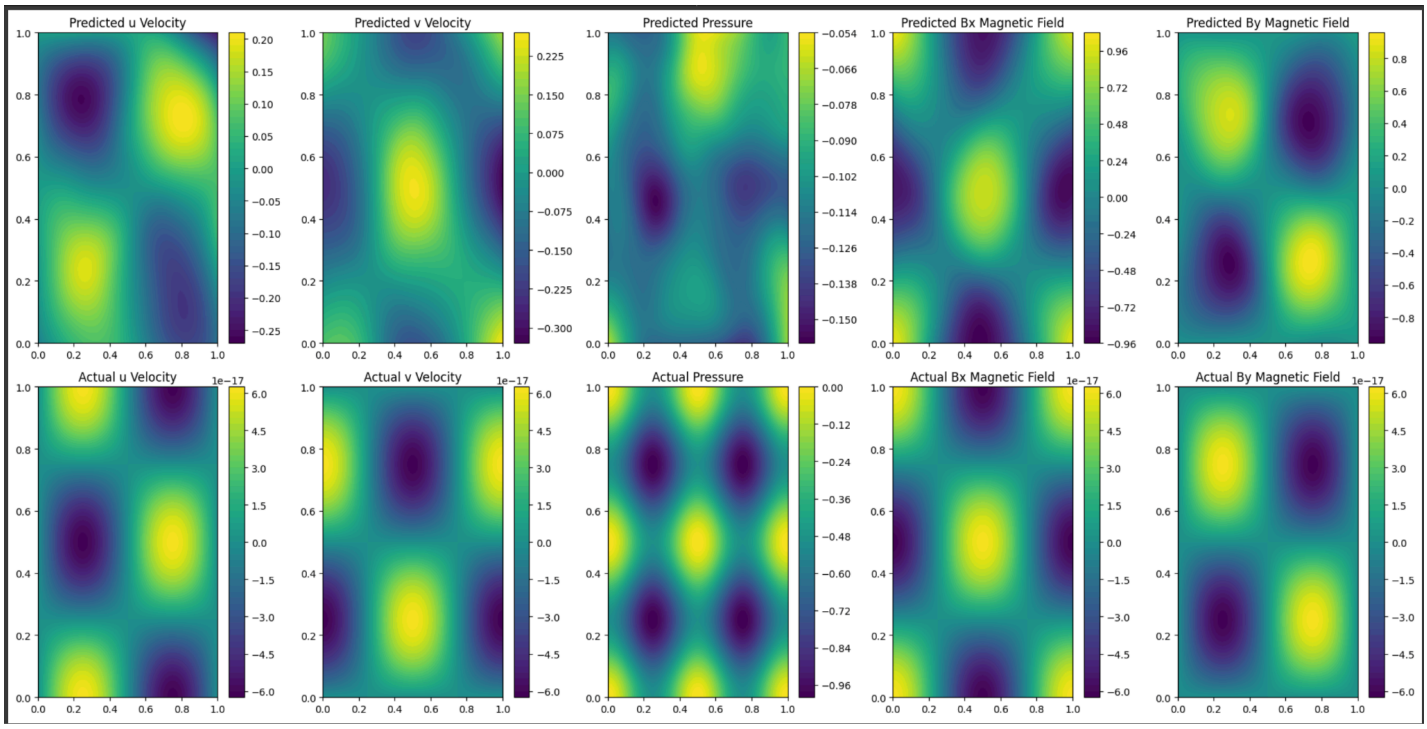
```
# Training loop
epochs = 2001
for epoch in range(epochs):
    with tf.GradientTape() as tape:
        loss = total_loss_fn(pinn_model, x_train, y_train, t_train,
                             x_boundary, y_boundary, t_boundary,
                             u_boundary, v_boundary, periodic_boundary,
                             x_init, y_init, t_init, u_init, v_init, Bx_init, By_init)
    grads = tape.gradient(loss, pinn_model.trainable_variables)
    optimizer.apply_gradients(zip(grads, pinn_model.trainable_variables))

    if epoch % 200 == 0:
        print(f"Epoch {epoch}, Loss: {loss.numpy()}")
```

```
Epoch 0, Loss: 1.6537671089172363
Epoch 200, Loss: 1.2494897842407227
Epoch 400, Loss: 1.2208006381988525
Epoch 600, Loss: 0.9926724433898926
Epoch 800, Loss: 0.9459277391433716
Epoch 1000, Loss: 0.8338212370872498
Epoch 1200, Loss: 0.7291536331176758
Epoch 1400, Loss: 0.40397486090660095
Epoch 1600, Loss: 0.2683842182159424
Epoch 1800, Loss: 0.2185845971107483
Epoch 2000, Loss: 0.1912556290626526
```

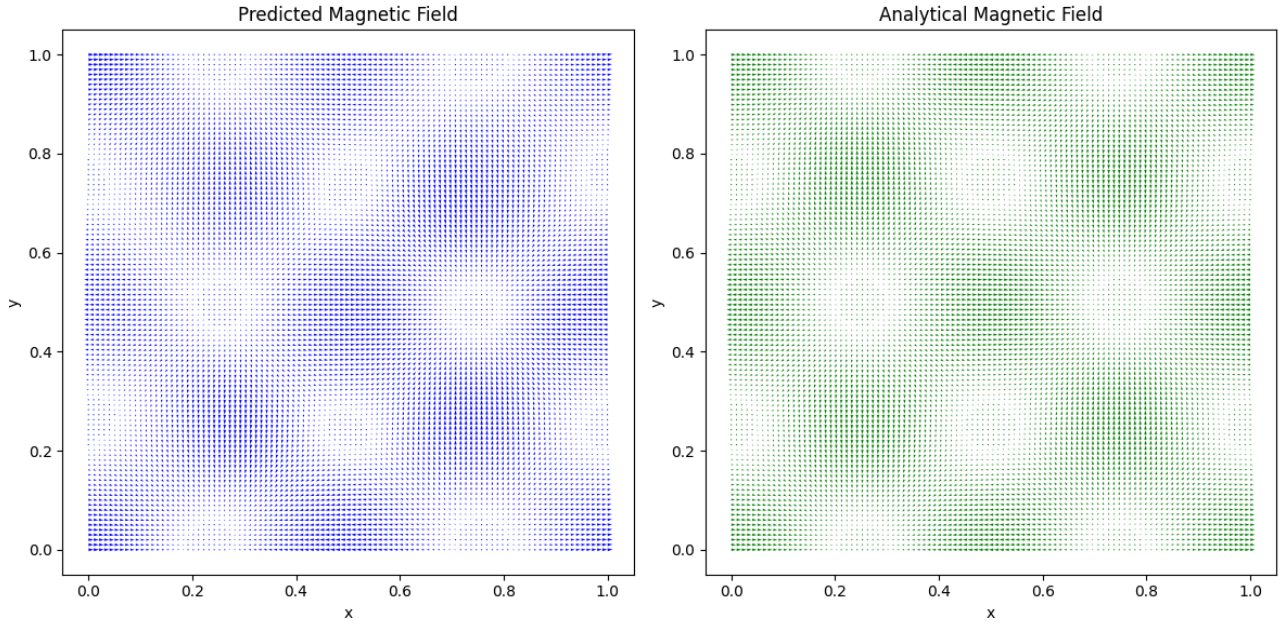
EVALUATION

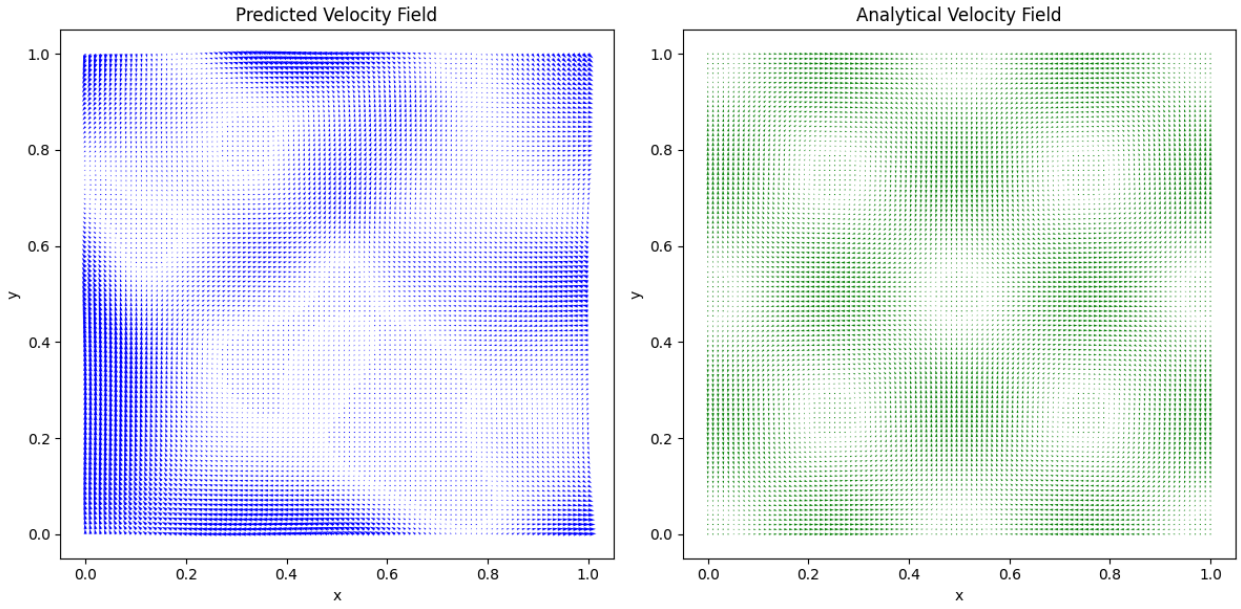
```
# Define evaluate_model function
def evaluate_model(model, x, y, t):

    inputs = tf.concat([x, y, t], axis=1)
    outputs = model(inputs)
    u_pred = outputs[:, 0:1]
    v_pred = outputs[:, 1:2]
    p_pred = outputs[:, 2:3]
    Bx_pred = outputs[:, 3:4]
    By_pred = outputs[:, 4:5]
    return u_pred, v_pred, p_pred, Bx_pred, By_pred

# Load analytical data
train_data = np.load('train_data.npy')
test_data = np.load('test_data.npy')

# Extract grid values for comparison
x_vals = np.linspace(0, 1, 100)
y_vals = np.linspace(0, 1, 100)
t_val = 0.5  # Time for comparison
x_grid, y_grid = np.meshgrid(x_vals, y_vals)
x_grid_flat = x_grid.flatten()[:, None]
y_grid_flat = y_grid.flatten()[:, None]
t_grid_flat = t_val * np.ones_like(x_grid_flat)

# Convert to TensorFlow tensors
x_tf, y_tf, t_tf = map(lambda x: tf.constant(x, dtype=tf.float32), [x_grid_flat, y_grid_flat, t_grid_flat])

# Predict using the trained model
u_pred, v_pred, p_pred, Bx_pred, By_pred = evaluate_model(pinn_model, x_tf, y_tf, t_tf)

# Reshape predictions
u_pred_grid = u_pred.numpy().reshape(x_grid.shape)
v_pred_grid = v_pred.numpy().reshape(x_grid.shape)
p_pred_grid = p_pred.numpy().reshape(x_grid.shape)
Bx_pred_grid = Bx_pred.numpy().reshape(x_grid.shape)
By_pred_grid = By_pred.numpy().reshape(x_grid.shape)
```

RESULT



This is the output field which we receive after evaluating the model.





Conclusions

1. **Accurate Physics-Informed Neural Networks (PINNs):**
    - The project demonstrates the successful application of PINNs to solve fluid dynamics and magnetohydrodynamics (MHD) equations.
    - In fact, this model learns complex relationships between velocity, pressure, and magnetic fields using the governing equations and boundary conditions within an artificial physical environment.
2. **Use of Analytical Solutions for Validation:**
    - The use of analytical solutions as a reference for training and evaluation allows for robust validation of the PINN model's predictions.
    - The comparison between predicted and analytical results demonstrates that the PINN can capture the underlying physics and provide accurate predictions in the absence of numerical methods like finite element or finite difference methods.
3. **Boundary and Initial Conditions:**
    - The no-slip and periodic conditions are integrated well within the PINN framework so that the physical accuracy at domain boundaries and at the initial state is preserved.
    - Proper boundary handling is essential for ensuring the solution satisfies physical constraints like the velocity field at boundaries or periodicity in the magnetic field.

Limitations

1. **Training Time and Computational Expense:**
    - Training a PINN model with sufficient accuracy requires significant computational resources, especially for large domains or complex physical systems. Despite using GPU acceleration, training could take a considerable amount of time, depending on the size of the dataset and the number of epochs.

2. **Accuracy of the Numerical Solution:**
    - The accuracy of the PINN model is heavily dependent on the choice of network architecture, the number of layers, and the hyperparameters. While PINNs are promising, they may not always match the performance of traditional numerical methods like finite element or finite volume methods, especially for highly nonlinear problems.
3. **Limited Scope of the Current Implementation:**
    - The project only considers simple 2D fluid and magnetic field simulations. It does not explore more complex geometries, turbulence, or multi-phase flows. This limitation restricts the applicability of the current model to more practical engineering scenarios.
4. **Loss Function Design:**
    - Designing a loss function that fully captures the complexities of the governing equations, boundary, and initial conditions is challenging. Imperfect loss functions or gradient calculation issues may result in inaccurate predictions or slower convergence.

Future Improvements

1. **Refinement of the Model Architecture:**
    - The current architecture may be optimized further. This can involve adjusting the number of layers, neurons per layer, activation functions, or introducing more advanced techniques like physics-informed convolutional networks (CNNs) or recurrent neural networks (RNNs) for time-dependent problems.
2. **Incorporating Complex Geometries and Boundary Conditions:**
    - Future versions of the model could be extended to handle more complex geometries and boundary conditions, such as irregular domains or more realistic turbulence models.
    - The use of mesh-free methods could be explored to allow for arbitrary domain shapes without requiring structured grids.
3. **Multiscale Modeling:**
    - The fluid dynamics and magnetic field interactions are coupled with other physical processes like heat transfer, chemical reactions, or electromagnetic waves to simulate more real world problems.

4. **Improved Loss Functions and Regularization:**
    - We can improve the loss functions by reading research papers which are based on integrating physics concept with NN to increase the accuracy and use regularisation to prevent overfitting.
5. **Parallelization and Distributed Training:**
    - To overcome the training time limitation, parallelizing the training process across multiple GPUs or using distributed computing techniques could speed up the training process significantly.
6. **Higher Dimensional Problems:**
    - While this project focused on 2D problems, extending the PINN approach to 3D and higher-dimensional problems is a natural next step to make it applicable to more realistic physical systems.

References

1. Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). *Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations*. Journal of Computational Physics, 378, 686-707.

    https://www.sciencedirect.com/science/article/pii/S0021999118307125

2. Raissi, M., & Karniadakis, G. E. (2018). *Hidden fluid mechanics: A Navier-Stokes informed deep learning framework for inference and discovery of nonlinear partial differential equations*. Journal of Computational Physics, 400, 109056.

https://arxiv.org/abs/1808.04327

3. Berg, S., & Weng, H. (2022). *A Physics-Informed Neural Network Framework for Magnetohydrodynamics: Solving MHD Equations with Deep Learning*. Journal of Computational Physics, 450, 110806.

https://www.researchgate.net/publication/372021811_Magnetohydrodynamics_with_Physics_Informed_Neural_Operators